CS 454 Final Project Report

Nicolas Randazzo

Tony Rago

17 December, 2023

Problem selected: 6

Given two strings x and y of the same length N over a finite alphabet Σ of size M. (N and M are user inputs so your solution should work for arbitrary N and M.) Consider a game involving two players A and B. A random string over the alphabet Σ is generated until x or y appears. (You can assume that x and y do not have a common prefix (other than the null string.) If the game terminates with the appearance of x (y), player 1 (2) wins. You are to write a program that takes as input the strings x and y, and outputs the probability that A wins.

Input:

Our program will prompt the user five times for input. The inputs are the length of the alphabet (M), the alphabet (Σ), the length of the strings (N), string x, and string y. First, Enter an integer that represents the length of the alphabet. Second, When entering the alphabet, separate each character by exactly one space. The alphabet must consist of only alphanumeric characters where no character is repeated twice. If the length of the entered alphabet does not equal M then you will be re-prompted. Third, enter an integer to represent the length of the strings x and y. Next, enter string x. Finally, enter string y. Both strings must only be made up of the characters specified in the alphabet. Also the strings must be the exact length that was specified previously.

Output:

The output of our program is one line describing the probability that Player A (string x) won the game.

Algorithm Used:

The DFA probability function takes as input the set of states Q, the alphabet Sigma, the transition function Delta, the initial state “0”, and the final state F (i.e. the last state). To generate the probability matrix, for each state a row is created in a matrix. For each state S in it’s outgoing

transitions 1/T (where T is the total number of outgoing transitions) is subtracted from each index of the row respective to the transitioning symbol. After the matrix has been created, the probability is calculated by taking the inverse of the probability matrix and multiplying it by b (where b is a vector containing all 0's except for the last index, which is 1). This returns a resulting matrix that is the product of the probability matrix and vector b.
As for time complexity, the **calculateProbability** method utilizes a numpy library function that inverts a probability matrix, which along with the standard matrix inversion algorithm, is **O(N^3)**. The **generateStates** method has a time complexity of **O(N)**, where N is the length of strings x and y. During transition matrix generation in **generateDelta**, the loop for generating the path for y and x is **O(N)**, while the loops in crissCrossTransitions are nested loops with complexity **O(N*M)**, where M is the size of the alphabet. During matrix computation in **computeMatrix**, the method loops over the list of states, and for each state, there is a loop over the alphabet. So, this method has a time complexity of **O(N*M)**, where N is the number of states and M is the size of the alphabet. The main function calls all of these methods, but the time complexity is dominated by the **calculateProbabilty** method.

Steps as pseudo code:

```
Input sigma
Input n
Input x
Input y
// len(x) == len(y) == n
DFAObject DFA
DFA.GenerateStates(x,y)
        // For len(x)+len(y)+1:
        //        Create new state
DFA.GenerateDelta(x,y)
        // for i in len(y):
        //        transition from state 0 to state 1:len(y)-1 on y[i]
        // repeat above for x starting at len(x)
```

ProbabilityMatrix = DFA.ComputeMatrix()

// Initialize an empty list matrix to store the rows of the matrix.

// Iterate over each state in the DFA's transition function (self.delta).

// Create an array arr filled with zeros, and set the value at the current state to 1.

// Check if the current state is an accepting or rejecting state. If true, append arr to matrix and continue to the next iteration.

// For each symbol in the alphabet, update the array based on transitions in the DFA's delta function.

// Append the updated array (arr) to the matrix.

// Convert the list of lists (matrix) to a NumPy matrix.

// Return the resulting matrix.

GameMatrix = DFA.CalculateProbability(ProbabilityMatrix)

// Initialize an array b of zeros with the length of the set of states (self.Q).

// Set the last element of b to 1.

// Convert the list b to a NumPy array.

// Compute the inverse of the matrix matrix using NumPy.

// Perform matrix multiplication between the inverse matrix (invMatrix) and the array b.

// Return the resulting matrix (result_matrix).

Calculate probability by subtracting 1 from the [0,0] of GameMatrix

Print(probability of win)

Data structures used:

Lists - Used to represent the returning Q (states of the DFA), F (accepting states of the DFA).

Dictionary - Used to represent the delta function for the DFA and minimized DFA.

Numpy Arrays: Used to represent the transition matrix during probability calculations.

Numpy Matrix: Used to calculate the probability matrix and result matrix.

Sample inputs & outputs

Example 1:

Enter alphabet: 0 1 2

Enter the length of the strings x and y: 3

X: 000

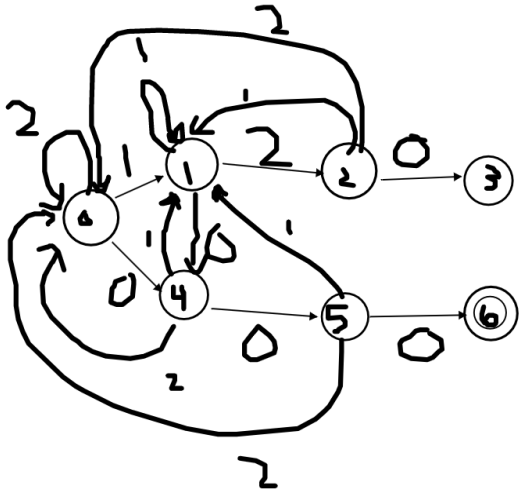
Y: 120

Probability that Player A wins: 61.9048% or: 0.61904761904762

```
Enter alphabet: 0 1 2
Enter the length of the strings x and y: 3
X: 000
Y: 120


Probability that Player A wins: 61.9048% or: 0.61904761904762
```

Example 1 DFA:



Example 2:

Enter alphabet: 0 1

Enter the length of the strings x and y: 4

X: 1011

Y: 0100

```
Enter alphabet: 0 1
Enter the length of the strings x and y: 4
X: 1011
Y: 0100


Probability that Player A wins: 35.7143% or: 0.3571428571428572
```

Example 3:

Enter alphabet: a b c

Enter the length of the strings x and y: 5

X: aaccb

Y: bbaca

Probability that Player A wins: 11.7971% or: 0.11797133406835858

```
Enter alphabet: a b c
Enter the length of the strings x and y: 5
X: aaccb
Y: bbaca


Probability that Player A wins: 11.7971% or: 0.11797133406835858
```

Conclusion:

Following the algorithm outlined in class, we were able to create functions that calculated the probability of a specific outcome from a DFA given user input. We found that this algorithm had minimum time complexity of $O(N^3)$ due to the matrix inversion function that was required for the end result. Pythons built in data structures proved extremely useful for the DFA class we created; lists containing the set of states, a dictionary containing key-value pairs of state-dictionary {transition: result_state} for the transitions, numpy arrays for holding the probabilities of each state on each transition, and numpy matrices for calculating the total probabilities of the DFA.

Sources:

Dr. Ravikumar - Algorithm from CS 454, Analysis from private meeting